GoDB Lab 4

For Lab 4, you will extend GoDB with one of the ideas we've discussed in class. You will be responsible for implementing and testing your code, and will provide a final writeup where you describe your approach and implementation, as well as any evaluation you did of your code.

As with previous labs you may work in teams of two for this lab.

We provide a brief description of several suggested below. For each project we provide ideas about the implementation and evaluation. We also indicate whether the project is "easy", "normal", or "hard". We will limit "easy" projects to a maximum grade of 90 points out of 100 and will give extra credit for hard projects.

If you would like to do a different project within GoDB, that is OK, but please check with us first to see if it is at the right level of difficulty.

If you are not sure which project to pick, we recommend either Column Store or Batch Iterator.

Project Ideas

1. **Column Store ("Normal")**

In this project you will modify GoDB to use a column-oriented physical layout. You should review Lecture 9.

The most straightforward approach is a so-called "early materialization" design where you construct complete tuples as you read columns off disk and process them with your existing query operators.

Recommended approach: Create a new DBFile interface that supports column-oriented files (call it "ColumnFile"). The ColumnFile iterator should accept a list of columns to read, and return tuples that contain only records from these columns. Physically, each table will now be implemented with multiple files, one per column. Each column will be split into a number of pages that should be managed by the buffer pool, as with heap pages, and your iterator should use BufferPool.GetPage as it iterates through these columns.

You will need a way to write out column files; you probably want to create something like HeapFile.LoadFromCSV. You will have to think carefully about how to support inserts and deletes in this scheme.

You should create test cases to verify that your approach can create column files and iterate over a subset of columns and that the GoDB operators like filter and join work properly in conjunction with your new DBFile.

Finally, you should modify the Parser to use your new ColumnFile implementation instead of HeapFiles. Parser is very complicated but because you are just swapping ColumnFile for HeapFile, this shouldn't be too complex. You will need to figure out how to determine the subset of columns to pass into the ColumnFile from the parsed query; we can help understand how the parser works if you are struggling with this.

At this point you should be able to use the GoDB shell to write queries over your column-oriented GoDB.

You should come up with an evaluation that shows that you get a performance gain from column stores; consider loading a large file with many columns and just iterating over a few of them.

Hard Variant: Extend your approach to use a "late-materialization" design. In this implementation, operators will pass around columns instead of tuples, and filters will produce bitmaps that indicate the positions in columns that are filtered out. Getting this to work requires many changes, including new variants of operators, rethinking iterators, and more.

2. **B+Tree - Read-only (easy) vs supporting full writes (hard)**

The goal of this project is to implement a B+ tree index for efficient lookups and range scans. We recommend reviewing Lecture 7 as well as Chapter 2 in Database Internals by Alex Petrov, which provides detailed information about the structure of B+ trees.

As discussed in class, the internal nodes in B+ trees contain multiple entries, each consisting of a key value and a left and a right child pointer. Adjacent keys share a child pointer, so internal nodes containing m keys have m+1 child pointers. Leaf nodes can either contain data entries or pointers to data entries in other database files. For simplicity, we recommend implementing a B+tree in which the leaf pages actually contain the data entries. Adjacent leaf pages are linked together with right and left sibling pointers, so range scans only require one initial search through the root and internal nodes to find the first leaf page. Subsequent leaf pages are found by following right (or left) sibling pointers.

Recommended Approach: Create a new DBFile interface for the B+ tree (call it "BTreeFile"). Unlike the HeapFile, the BTreeFile consists of four different kinds of pages: two different kinds of pages for the internal and leaf nodes of the tree (“BTreeInternalPage” and “BTreeLeafPage”), header pages (“BTreeHeaderPage”), and one page at the beginning of every BTreeFile which points to the root page of the tree and the first header page (“BTreeRootPtrPage”).

For the read-only BTreeFile, you have to implement these DBfile interface methods: readPage(), pageKey(), Descriptor(), and Iterator(). You also have to implement an additional method that finds a leaf page by key (called “findLeafPage()”). Having this method could help your implementation of readPage(), and you should create test cases to verify that your approach finds the correct leaf page given the key. As for Iterator(), we recommend performing an implicit DFS on your B+ tree by having the root and internal pages iterators recursively call children iterators and have the leaf page iterate through its tuples.

Hard Variant: To support full writes, you have to implement these remaining DBfile interface methods: insertTuple(), deleteTuple(), and flushPage().

For insertTuple(), findLeafPage() can be used to find the correct leaf page into which we should insert the tuple. If the leaf page is full, attempting to insert a tuple into the full leaf page should cause that page to split so that the tuples are evenly distributed between the two new pages. Each time a leaf page splits, a

new entry corresponding to the first tuple in the second page will need to be added to the parent node. Occasionally, the internal node may also be full and unable to accept new entries. In that case, the parent should split and add a new entry to its parent. This may cause recursive splits and ultimately the creation of a new root node.

For deleteTuple(), attempting to delete a tuple from a leaf page that is less than half full should cause that page to either steal tuples from one of its siblings or merge with one of its siblings. If one of the page's siblings has tuples to spare, the tuples should be evenly distributed between the two pages, and the parent's entry should be updated accordingly. However, if the sibling is also at minimum occupancy, then the two pages should merge and the entry deleted from the parent. In turn, deleting an entry from the parent may cause the parent to become less than half full. In this case, the parent should steal entries from its siblings or merge with a sibling. This may cause recursive merges or even deletion of the root node if the last entry is deleted from the root node.

Warning: as the B+Tree is a complex data structure, it is helpful to understand the properties necessary of every legal B+Tree before modifying it. Here is an informal list:

1. If a parent node points to a child node, the child nodes must point back to those same parents.
2. If a leaf node points to a right sibling, then the right sibling points back to that leaf node as a left sibling.
3. The first and last leaves must point to null left and right siblings respectively.
4. Record Id's must match the page they are actually in.
5. A key in a node with non-leaf children must be larger than any key in the left child, and smaller than any key in the right child.
6. A key in a node with leaf children must be larger or equal than any key in the left child, and smaller or equal than any key in the right child.
7. A node has either all non-leaf children, or all leaf children.
8. A non-root node cannot be less than half full.

We expect you to write some tests that check these properties in the face of some insertions, deletions, and some interleaving of them.

3. Optimizer + Stats (normal)

The goal here is to implement a fully functional query optimizer for GoDB, including a plan enumerator, a cardinality estimator, and a cost model. You will have the freedom to explore different kinds of plan enumeration methods and cardinality estimators, but we provide the following recommendations similar to the Selinger method covered in lectures.

Recommend Approach:

- **Table statistics collection:** We recommend using equal-width histograms to maintain a set of statistics for each column of each table (as we discussed in lecture 14). Specifically, you can define a struct called IntHistogram with two important fields: bin_breaks and density.
    - To make your implementation easier, you can only implement the integer histograms and assume that all the columns in your database have integer type. Then, you will need to scan (a sample of) each table in your database and build one IntHistogram for each column.
    - In addition, you may also want to maintain the stats on the number of unique values for each column (or only the join keys) for the purpose of estimating the cardinality of joins.

    - You may implement slightly more sophisticated stats, such as equal-depth histograms and most-common-value tables.
    - Optionally, you can implement mechanisms to update the collected statistics after your underlying data changes.
- **Single-table cardinality estimation:** You should implement a cardinality estimator using the table stats you collected. Although you can implement more sophisticated methods, we highly recommend you adopt the attribute independence assumption as described in lectures. Specifically, you can use your IntHistogram to estimate the selectivity of the filter on each filtered column and multiply the estimated selectivities together on all columns.
- **Join cardinality estimation:** We recommend you adopt the join-key uniformity assumption as described in lectures. Specifically, you should first get the estimated base-table cardinality and then use the collected statistics on the total number of unique values on the join keys to estimate the join size.
    - To make your implementation easier, your implementation only needs to support inner join and equality join.
    - You may implement slightly more sophisticated methods, such as the joining histograms method described in lecture 14.
- **Cost model:** It is optional to have a fully-functioned cost model in this task. This is because the cost model helps select the best physical operator (i.e., access methods and join algorithms), but most of you only implemented one access method (sequential scan) and one join algorithm in lab2. However, if you have implemented multiple physical operators, we recommend you implement the cost model as described in Lectures 6&7. Otherwise, your cost model only needs to distinguish which table (or intermediate join result) to put as a left child or right child of a join.
- **Join Ordering:** We recommend you use the Seligner dynamic programming algorithm to enumerate the query plan and select the best join order. To make your implementation easier, we provide the following pseudocode:

```
j = set of join nodes
for (i in 1...|j|):
    for s in {all length i subsets of j}
      bestPlan = {}
      for s' in {all length d-1 subsets of s}
          subplan = optjoin(s')
          plan = best way to join (s-s') to subplan
          if (cost(plan) < cost(bestPlan))
             bestPlan = plan
     optjoin(s) = bestPlan
 return optjoin(j)
```

- **Overall structure:**
    - You should implement a function resembling an “ANALYZE” command to collect all necessary statistics. You should execute this function right after all data has been loaded to your database and before query execution.
    - We have provided the parser for you that you have already played with in lab2. In parser.go file, line 1294 we parse the query statement into a logical plan, containing a list of tables and different operators. You will need to modify the makePhysicalPlan function in parser.go. Specifically, the current version of makePhysicalPlan function we provided with you generates a fixed join order. You need to modify the for loop starting in line 871 to call your join ordering function. You may need to change other places of this function.

As a deliverable, we expect you to write your own test cases, testing the collected table stats, the accuracy of the cardinality estimator, the cost model, and the plan enumerator. You should also verify that your modified GoDB can execute queries (e.g. TestParseEasy in lab2). In the end, we expect you to test on your own database (this could be MBTA as we provided in lab2) and your own choice of queries. You should report the performance difference by comparing your query optimizer implementation with the original GoDB (without a query optimizer).

4. **Recovery (normal)**

The goal here is to implement log based recovery, relaxing the FORCE / NO STEAL assumption and using a log to recover the system back into a transaction consistent state when a crash happens.

Recommend Approach: You are free to try to implement logging however you would like, although we recommend not trying to implement the full generality of ARIES. In particular, we recommend doing whole-page physical UNDO (while ARIES must do logical UNDO) because we are doing page level locking and because we have no indices which may have a different structure at UNDO time than when the log was initially written. The reason page-level locking simplifies things is that if a transaction modified a page, it must have had an exclusive lock on it, which means no other transaction was concurrently modifying it, so we can UNDO changes to it by just overwriting the whole page. For a similar reason you can do whole page physical REDO.

One basic approach for your implementation would be:

- Modify your GoDB code to allow dirty pages to be flushed to disk. Write test cases that show that you can modify more pages than you have in your buffer pool.
- Implement a log data structure / file format. Write test cases to verify that your log entries work as expected.
- Modify GoDB to pass data about the log file into the buffer pool, heap files, etc.
- Whenever a transaction starts, write an entry to the log indicating the transaction started. Write test cases to ensure this is done properly.
- Whenever a dirty page is flushed to disk, first write a log entry to the log. Ensure the log is flushed before the page is flushed (you may want to flush the log after every write.) Write test cases to ensure the log contains the expected data after a write / flush.
- Whenever a transaction commits or aborts, write an entry to the log indicating the outcome of the transaction. Write test cases to ensure that this is done properly.
- When a transaction aborts, play the log backwards from the last LSN of the aborting transaction, undoing modifications to pages it wrote (note that the aborting transaction must have been the only transaction modifying any pages it wrote, so undos can be done without worrying about concurrent modifications.) Write test cases to test this abort method – i.e., that after an abort modifications of transactions are successfully undone even if the aborting transaction flushed data to disk.
- When the system crashes and restarts, the system should use the log to recover. You can do UNDO and then REDO or REDO then UNDO. You will need to think about where each phase should start and end. Write test cases to verify that the overall recovery process works properly.

You will need to think about several things, including the format of log records, what data structures (like the ARIES transaction table) you keep in memory when the system runs, how to write checkpoints (if you choose to use them, etc.)

Do some evaluation of the overhead of adding recovery to your implementation (how much slower are writes?) and the recovery time of your system.

Hard Variant: Extend your implement to support compensation log records or record level locking.

5. **Mini-batch Iterator (normal) / Code Generation (hard)**

As we discussed in class the tuple-at-a-time iterator model is very inefficient. Modern database systems implemented a range of techniques to improve the performance of the iterator model.

Mini-batch Iterator (medium)

One common strategy is to use so called mini-batches. Instead of passing around a tuple-at-a-time the iterators pass around small batches (e.g., 1000 tuples) at a time. Doing so amortizes the overhead of the iterator model as each iterator can process many tuples at once tuples. More concretely, as part of this project you have to modify the iterator interface to something like:

```
func (joinOp *EqualityJoin[T]) Iterator(tid TransactionID) (func() ( []*Tuple, error), error)
```

This will require modifying the iterator implementation of all of the operators and the heap file. Each operator should try to fill up the batches as much as possible. You might have to make additional changes to if an array cannot be fully filled, if there are null batches, the end of an operator, etc. As part of your project you should also benchmark your new mini-batch iterator implementation against the original tuple-at-a-time implementation and modify the SQL parser to support this new iterator model.

IR Interpreter (hard)

Another common technique to speed-up query execution is to not use the iterator model at all and instead compile queries to an intermediate representation (IR), which is then either interpreted or compiled. For example, SQLite uses this approach and compiles everything down to an IR. Part of this project is to replace the iterator model entirely and replace it with an IR based query execution model.

For more information on IR and query compilation, see:
https://15721.courses.cs.cmu.edu/spring2020/papers/14-compilation/p539-neumann.pdf
https://db.in.tum.de/~kersten/Tidy%20Tuples%20and%20Flying%20Start%20Fast%20Compilation%20and%20Fast%20Execution%20of%20Relational%20Queries%20in%20Umbra.pdf?lang=de

6. **Parallel (normal to hard), Distributed (hard)**

Modern database systems allow for inter- and intra-query parallelism. That is, they not only support running several queries at the same time but also use multiple threats (i.e., compute cores) to process a single query to reduce the query latency (intra-query parallelism) . The goal of this project is to add intra-query parallelism using threads to GoDB. This requires parallelizing several operators, such as scans, filters, and joins. An “easy” implementation of intra-query parallelism is to assign each operator to a thread and use an intermediate result buffer. Within the threat you would implement a pull-based processing that tries to process the input tuples as quickly as possible in a loop while storing all the output results in a buffer. Each next() call on the operator returns then a tuple from the buffer. If no tuples are available, the next() call blocks until tuples become again available or there are no further tuples to

process. This parallel implementation of a query processor is not perfect though and might result in a lot of busy waiting and context switching. However, it does allow for a high degree of parallelism particularly for bushy query trees.

Hard Variant I: A better (and harder) implementation would leverage the above mini-batch iterator model. With mini-batches the overhead of the iterator and context switching can be avoided.

As part of the project, you need to evaluate the single vs multi-thread implementation. You are also expected to add sufficient test cases to show that the implementation is correct and does not create deadlocks among other potential multi-thread problems

Hard Variant II: Another hard extension is to make GoDB distributed. That is queries can use multiple machines for data processing. If you decide to build a distributed GoDB, we strongly suggest you focus on analytical processing (OLAP) rather than transactional processing. The latter requires implementing some form of distributed concurrency control, which is very very hard. But even for OLAP, you need to to develop a partitioning scheme (e.g., how to distribute data across machines) and at least a limited form of distributed query processing (e.g., distributed scans and filters).

7. **OCC -** Page level (normal), record level (hard)

In this project you will support using Optimistic Concurrency Control for transactions, which is covered in lecture 11. The goal here is to add another method for concurrency control besides the 2PL approach you implemented in lab 3.

Recommended approach: In this outline, we assume page-level locking but the approach can be applied analogously for record-level locking. First, you will need a data structure in the BufferPool that keeps track of the running transactions, together with the phase they are currently in (e.g. read phase, validation phase, write phase). Second, you will need a data structure in the buffer pool that keeps copies of the pages a transaction uses. When a transaction accesses a page, these accesses should happen on the transaction's page copy (e.g. calls to BufferPool.GetPage should return the transaction's copy of the page). Finally, for validation, you need a data structure in the buffer pool that keeps track for each transaction, which other transactions were concurrently accessing the same page and what kind of access that was. Think how you can implement such a data structure. Note that it's not sufficient to only have a map recording which tid accessed which page with what permission: After a transaction finished, you cannot yet remove it from this map since other transactions that enter the validation phase at a later point might need to know which pages it accessed to check for conflicts. Instead, if every transaction keeps its own record of conflicts, you can delete a transaction's conflict record once it is finished (i.e. aborted or committed). Let's refer to this final data structure as the "concurrent access record".

During the read phase, you need to keep track which transactions concurrently access the same pages in the concurrent access record. For example, if a transactions wants to write to a page that another transaction concurrently wants to read, the concurrent access record needs to allow transactions to check if they conflict with the other one during the validation phase (whether a transaction is aborted will then depend on when they finished their read and write phases).

You can initiate the validation and write phase in CommitTransaction. First, you should update the phase of the committing transaction in the corresponding data structure in BufferPool. Second, you should use the conditions covered in the lecture to validate if the transaction is allowed to write its changes back or

has to abort. If the transaction has no conflicts, you can write the modified page copies back to the buffer pool. Otherwise, you need to abort the transaction (i.e. discard its page copies). You need to think about how to let other transactions know whether the committing transaction successfully committed or was aborted, as other transactions might need this for validation. For example, when aborting, you could remove the transaction's read and written pages from the concurrent access record of other transactions.

At this point, you should be able to run transactions using OCC as you did using 2PL in lab 3. In your evaluation, you should come up with a workload where you show the benefit of OCC over 2PL and another workload where you show the benefit of 2PL over OCC.

Hard variant: Lock at tuple granularity.

How we will grade

Implementation - 25%
Test cases - 10%
Evaluation / validation - 15%
Writeup - 40%
Project Proposal - 10%

What you need to hand in

Project proposal/plan (1-2 pages) - You should describe the project you have chosen, and the basic plan of attack you plan to take. What parts of GoDB will you need to modify? What new test cases will you write? How will you evaluate your solution? How will you know if your project works / is successful? If you are working in a team of two, what will the division of labor be?

Final report (no set length, but no need to be overly verbose - you may reuse text from your proposal / plan) -
Your report should have the following:

- The project you chose
- Background about the problem and why it is helpful in GoDB
- A high level description of the approach, including the modifications you had to make to GoDB
- A detailed presentation of key features of your implementation, e.g., a walkthrough of essential pieces of your code
- A list of test case you wrote and what they test, and why you feel they are sufficient to demonstrate that your implementation works as expected
- Any evaluation you did, e.g., of the performance gains or effectiveness of your solution. You should run benchmarks on test data of appropriate size and queries of the appropriate type to show these benefits.
- A discussion of things that were challenging or required additional work
- A discussion of things that didn't work as you expected or that are still remaining to be done
- A link to your GitHub repo or code repository